ACEEE Int. J. on Information Technology, Vol. 3, No. 2, June 2013

Exploring the Solution Space of Sorting by Reversals: A New Approach

Amritanjali[1] and G. Sahoo[2]
[1] Department of Computer Science & Engineering, BIT Mesra, Ranchi, India
Email: amritanjali@bitmesra.ac.in
[2] Department of InformationTechnology, BIT Mesra, Ranchi, India
Email: gsahoo@bitmesra.ac.in

***Abstract*— Analysing genome rearrangements is a problem from the vast domain of comparative genomics and computational biology. Several studies have shown that closely related species have essentially the same set of genes however their gene orders differ. The differences in the gene order are the results of various large-scale evolutionary events of which reversal is the most common rearrangement event. The problem of finding the shortest sequence of reversals that can transform one genome into another is called the sorting by reversals problem. The length of such a sequence is the reversal distance between the two genomes. In comparative genomics, sorting by reversals algorithms are often used to propose evolutionary scenarios of large-scale genomic mutations between species. Following the first polynomial time solution of this problem, several improvements has been published on the subject. In 2008, Braga *et al.* proposed an algorithm to perform the enumeration of *traces* that sort a signed permutation by reversals. This algorithm has exponential complexity in both time and space. To efficiently handle the traces, Baudet and Dias proposed a depth first approach in 2010. However, one of the limitations of the proposed algorithm was that it cannot provide the count of number of solutions in each trace. In this paper we are presenting an algorithm to list the normal forms of each trace in depth first manner and provide count of the total number of solutions in the solution space.**

***Index Terms*— Comparative Genomics, Genome Rearrangements Algorithms, Sorting by Reversals**

I. Introduction

The appearance of gene content and gene order data has taken phylogenetic studies to a new level. The difference between the order and orientation of genes on a chromosome in two genomes can be used as a measure of evolutionary distance. Genomes evolve by some specific operations known as genome rearrangements. Genome rearrangements are large scale mutations that can alter not only the ordering and orientation (strandedness) of the genes on the chromosomes but also the genome content. As large scale mutations are rare events compared to point mutations, they can give us valuable information about ancient events that accompanied evolution of organisms. This is the main motivation behind studying genome rearrangements and finding the most "plausible" evolutionary scenario between two genomes. From parsimony hypothesis, the most plausible evolutionary scenario between any two genomes is described by an optimal sequence of rearrangements that transforms one genome into the other. In the classical approach, each gene has exactly one copy in each genome, and only operations that do not change the genome content are considered. Reversal (inversion) is the most common rearrangement operation. It reverses the order of a segment of genes in the chromosome, reversing their direction of transcription. We have to compare the given genomes with respect to the relative order of the genes in them as well as the relative direction of transcription. Henceforth, a sign (+ or -) is associated with each gene representing its transcriptional orientation, i.e. on which of the two complementary DNA strands the gene is located. So, the genomes are represented as signed permutations of genes. One of the genome is taken as the source genome and the other one as the target. The problem of finding the shortest sequence of reversals that can transform a source permutation into the target permutation is called as sorting by reversals problem. The length of such a sequence is the reversal distance between the two genomes. Based on the notion of breakpoint graph, Hannenhalli and Pevzner [1] developed a polynomial-time algorithm for sorting signed permutation by reversals that runs in $O(n^4)$ time for permutations of n genes, indicating the extent of genome rearrangement required. After many subsequent improvements on the running time currently, the best known algorithm for this problem runs in $O(n^{3/2}\sqrt{\log n})$ time [2], while the reversal distance can be computed in linear time [3].

The problem of sorting signed permutation by reversals has been the subject of several literatures. The proposed algorithms suggest one optimal solution. However, one study by Siepel [5] showed that for most of the permutations several minimum-length sequences of reversals exist. Knowing all minimum-length sequences provide more information, thus improving the usability of the result. For example, it will be useful in assessing the biological merits of various parsimonious rearrangement scenarios. Using the algorithm proposed by Siepel, the set of all optimal 1-sequences of reversals for a permutation can be calculated in $O(n^3)$ time. The problem was called as All Sorting by Reversals (ASR) problem. Recently, their methods have been improved to an average-case $O(n^2)$ algorithm [6]. The number of optimal sorting sequences is usually huge. Attempts were made to reduce the size of set of solutions by using some biological

DOI: 01.IJIT.3.2.1228

constraints, such as giving preference to small reversals [4]. To precisely present the set of solutions Bergeron et al. [7] provided a method to group the generated sequences of reversals into equivalence classes. Combining the results of Siepel and Bergeron et al., Braga et al. [9,10] developed an algorithm that outputs one representative element per class of solutions and provides the count of the number of sorting sequences in each class. With several examples they have shown that their result is more relevant than giving just one solution or listing all the solutions. Solution space is drastically reduced when dealing with traces of solutions; however, it is still too big to be handled by biologists on large permutations. Another drawback is its large time complexity that makes it run slower for large permutations. The algorithm implementation is limited to permutations with reversal distance bounded by 20 mainly because of memory. Baudet et al. [11] improved upon their work by generating the normal forms in an economical way; however, they cannot provide total count of solutions represented by each trace. In this paper we are present an algorithm to list the normal forms of each trace in depth first manner and provide count of the total number of solutions in the solution space.

This paper is an extended version of the previously published paper [12]. This paper is organized as follows. Section II gives the necessary background and formalizes the sorting by reversals problem. In section III we give details of our approach. Section IV illustrates the algorithm with an example. Section V concludes the paper and suggests possible improvements.

II. Background

This section provides the basic background for the mathematical analysis of genome rearrangements such as reversals. The classical approach for addressing the problem of sorting a signed permutation by reversals is based on the theory of breakpoint graph.

A. Genome Representation

In the model we consider unichromosomal genomes. A chromosome is a sequence of genes. We represent the studied genomes by a list of homologous markers (usually a gene or a sequence of genes). These homologous markers are represented by signed integers, assuming that all the markers are unique. The order and orientation of markers on one genome in respect to the other is represented by a signed permutation $\pi = (\pi_1 \dots \pi_n)$, where n is number of homologous markers. Reversals may change the order and orientation of some of the markers, by reversing a segment of the genome and also the DNA strand the segment is on. The evolutionary distance between two species is estimated by the minimum number of rearrangement operations required to transform the permutation representing the genome of one into that of the other. Here we are considering only reversal operation, so this is known as reversal distance between the permutations.

In order to find the optimal sequence of reversals, the target genome is represented by an identity permutation (1 … n) and is denoted by *Id*. The other genome is represented by a signed permutation $\pi = (\pi_1 \dots \pi_n)$ over (-n, …, -1,1, … , n) showing the order and orientation of each marker relative to the target permutation. Here, π_i is the element at position i in the permutation π. Now, we have to transform the signed permutation π into the identity permutation *Id* through an optimal sequence of reversal operations. Thus, the problem is known as Sorting by Reversals.

B. Solution Space of Sorting by Reversals

Let $\pi = (\pi_1 \pi_2 \dots \pi_n)$ be the source permutation of size n, where each element in the set {1, 2, 3, … , n} has either plus or minus sign. A reversal $\rho(i, j)$ on an interval $[i, j]$ of π is the transformation-

$$\pi.\rho = (\pi_1 \dots \pi_{i-1} - \pi_j \dots - \pi_{i+1} - \pi_i \pi_{j+1} \dots \pi_n) \quad (1)$$

The resulting permutation, denoted as $\pi.\rho$, is the permutation obtained from π by reversing the order and flipping the signs of the elements in the interval $[i, j]$. If $\rho_1,\dots,\rho_k$ is a sequence of reversals, we say that it sorts a permutation π, iff $\pi.\rho_1 \dots \rho_k = Id$. A shortest sequence of reversals sorting a permutation is called an optimal sorting sequence. The length of such sequence of reversals is called reversal distance, $d(\pi)$. We are assuming that for the given permutation there exists at least one sequence of reversals that sorts the permutation, called permutation without hurdles. Hurdles are very rare, both in random permutations and in permutations that are associated to biological data [7]. Permutations that have no negative elements are examples of permutation with hurdle.

The traditional algorithms for sorting by reversals problem output just one optimal solution for the problem, while there can be large number of optimal solutions. In 2002, the work done by Siepel [5] resulted in a method to enumerate all solutions. Using the algorithm proposed by Siepel, we can find out the set of reversals that will bring the given permutation one step closer to the target. A sequence of reversal $s = \rho_1 \rho_2 \dots \rho_i$ is called an opimal i-sequence if $d(\pi. \rho_1 \rho_2 \dots \rho_i) = d(\pi) - i$. And, s is an optimal sorting sequence iff $i = d(\pi)$. Using the Siepel's algorithm the set of all optimal 1-sequences of a permutation can be calculated in $O(n^3)$ time. Taking each reversal ρ of this set and applying it over the original permutation π, we obtain a new set of permutations. Each permutation π' of this set has reversal distance $d(\pi') = d(\pi) - 1$. When we apply Siepel's algorithm over π', we get a new set of optimal 1-sequences. If we combine these 1-sequences with its predecessor ρ, we get a set of optimal 2-sequences. Therefore, by iterating this algorithm, we can obtain the set of all optimal $d(\pi)$-sequences that sort the permutation π.

However, the size of the solution space was huge and the number of sorting sequences of reversals increases exponentially with the size of the permutation and the reversal distance. For example the permutation (-4 -3 12 -11 -8 10 9 7 -6 -5 2 -1) has 31,752 optimal solutions. The huge size of the solution space was obstacle for the result to be used

DOI: 01.IJIT.3.2.1228

practically. Bergeron et al. [7] provided a more relevant result to the problem of sorting by reversals by presenting a method to group the generated sequences of reversals into equivalence classes based on commuting properties of reversals. Two consecutive reversals α and β in an optimal sequence of reversals are said to commute if they do not overlap, i.e. either their intervals are disjoint or one is completely contained in the other. The order in which these reversals are applied does not matter, they will have same effect. So, the sequence of reversals $\alpha\beta$ is equivalent to $\beta\alpha$. All the sorting sequences of reversals that consist of same set of reversals constitute one equivalence class because they can be transformed into each other by interchanging the commuting reversals. Consider for example the permutation π = (1 -5 4 -3 2). The sequence s = {2}{2, 3,4,5}{4} is an optimal sorting sequence. Any permutation of these three reversals is also an optimal sorting sequence, as they are all commuting with each other. All of these sorting sequences are equivalent and belong to same equivalence class. The concept of traces [8] is used to represent the equivalence class by a normal form. For any trace, there is a unique element that is in normal form.

A trace T is a set of equivalent words over an alphabet A. An element s of a trace T is in its normal form if it can be decomposed into subwords $s = u_1 | \ldots | u_m$ such that:

- every pair of elements of a subword u_i commute;
- for every element α of a subword u_i (i >1), there is at least one element β of the subword u_{i-1} such that α and β do not commute;
- every subword u_i is a nonempty increasing word under the lexicographic order induced by A.

For example the permutation π = (1 -5 4 -3 2), there is only one trace given by {2}{2,3,4,5}{4} which is in normal form. It has only one subword with three elements. Consider another example permutation π = (-3 2 1 -4) that has two traces of optimal sequences of reversals, {1} {1,2,3} {2} {4} and {1,2,4} {3} | {1,3,4}| {2,3,4}. In the second trace the reversals {1,2,4}, {1,3,4} and {2,3,4} overlap with each other resulting in three subwords. Only reversal {3} commutes with all the other reversals, so by interchanging {3} with others we get 4 solutions for this trace.

The algorithm developed by Braga et al. [9, 10] is capable of enumerating the set of all traces that represents all possible solutions. It combines the work of Seipel with the concept of traces. The algorithm first calculates the set of 1-traces. Then, at each subsequent step i ($\leq d(\pi)$) every element in the set of (i -1)-traces is used as prefix to build an i-trace. To construct the normal form of a trace, a reversal is added in the lexicographic order to the subword following the last subword with which it doesn't commutes. If the reversal does not commute with the last subword then a new subword is added with it. If it commutes with all the subwords then it is added to the first subword. The implementation of their algorithm [9] was shown to run much faster than previously published methods. However, it is still limited to small permutations mainly because of memory. The extensive use of memory is due to the fact that, in order to create (i+1)-traces, we have to store all the i-traces as they are used as prefix. Each time an (i+1)-trace is created it is compared with other traces to see if it is a new trace or not. In 2010, the work done by Baudet et al. [11] allows us to generate normal forms in depth-first manner as opposed to the inherently breadth-first approach of Braga. As a result, their algorithm eliminates the need for a potentially exponential amount of memory. However they failed to provide count of the number of solutions in the traces. Motivated by the work done by Baudet we present an algorithm to list the normal forms of traces in depth first manner and provide the total number of solutions in the solution space. We, also, provide an approach for implementing the algorithm in parallel.

III. Proposed Work

Baudet showed that in order to generate all (i+1)-traces we do not actually need to store all i-traces. We can explore each branch separately starting from each reversal in the optimal 1-sequences of reversals of the input permutation. The set of optimal 1-sequences of reversals is processed in lexicographic order. Based on this we proposed a parallel solution for listing the normal forms of the traces [12]. Each branch in the solution space is explored by a separate process. This results in better time and space complexity. One of the limitations of the proposed algorithm is that it works when there is at least one solution in each trace which is in lexicographic order. To create (i+1) trace we calculate the next set of optimal 1-sequences of reversals from the i-traces in the branch being explored. However, not all the reversals from this set are in normal form order when they are appended to the i-trace. Only those that are in normal form order are explored further, others are rejected. From the definition of the normal form we can identify if the next reversal, when appended to an i-trace, will be the last reversal of (i+1)-trace or not. Let S_k be the last subword of an i-trace and ρ_i is the last reversal of S_k. A reversal ρ_j can become the last reversal of the corresponding (i+1)-trace only if they satisfy any one of these two conditions –

1. The reversal ρ_j does not commute with the subword S_k.
2. The reversal ρ_j commutes with S_k ($k \geq 1$)but does not commute with S_k-1 ($k \geq 1$) and is lexicographically bigger than ρ_i.

In other cases we do not need to store the reversals as either they are already processed or they will be processed later where it will be in appropriate order. Using this concept, we present the modified algorithm for listing the normal form of traces.

A. Enumeration of Traces

First of all we find the set of optimal 1-sequences of the reversal using Siepel's algorithm [5]. The set of next reversals consists of commuting and non-commuting reversals that can be added to the i-trace to create the (i+1)-traces. Only those reversals are explored further that satisfy one of the above two conditions. We call them as *valid* reversals. We are using the tree structure given in [12] where a node in the tree is represented by an array of a *rNode* structure. However,

DOI: 01.IJIT.3.2.1228

we have dropped the field count (C) from the rNode structure. So, it has following 3 fields:

1. *reversal* (R)- a reversal interval of the optimal 1-sequences
2. *permutation* (P)- permutation obtained after applying the *reversal*
3. *next*- points to the array of *rNode*s that contains the optimal 1-sequences obtained from *permutation*

In [12] the field count(C) was used for traversing after its expansion was complete to list the traces found. Instead of doing this, here we are keeping the concatenated list of reversals from the root node to the current node. This is used not only in printing the trace when we reach the last level but also in determining if the next reversal is valid or not.

Algorithm 1 *listTraces(* π*)*

```
begin
    d ⟵ reversal distance of π
    R ⟵ {ρ | ρ is an optimal 1-sequence for π in
                lexicographic order}
    N ⟵ count(R) [number of elements in R]
    First ⟵ allocate(N)
    for j ⟵ 0 to N-1 do
        [initialize rNode fields]
        initialize(First + j, ρ_j, π, null)
        [generate all the subsequent (d-1)-sequences]
        expand (First +j, d-1, ρ_j)
    end for
end
```

The reversals in the root node (node at the level-1, First) are stored in lexicographic order. The routine *listTraces* (Algorithm 1) initializes the root node (First) and calls the recursive routine *expand* (Algorithm 2) to explore each path originating from here. We have added one parameter P (prefix) to the *expand* routine for storing the concatenated list of reversals in the normal form order in the path from the root node to the current node.

Algorithm 2 *expand*(current, d, P)

```
begin
    current.P ⟵ applyReversal(current.R, current.P)
    if (d=0) then
      current.next ⟵null
      printTrace(P)
    else
      R ⟵ {ρ | ρ is an optimal 1-sequence for
    current.P in lexicographic order}
     N ⟵count(R) [number of elements in R]
      B ⟵ allocate(N)
      current.next ⟵ B
      for j ⟵0 to N -1 do
        [initialize rNode fields]
        initialize((B + j), ρ_j, current.P, null)
        [generate all the subsequent (d-1)-
                                sequences]
        if valid(ρ_j, P) then
            [append ρ_j to P and explore
    further]
            (B+j).w⟵ expand ((B + j), d-1, P.ρ_j)
        endif
      end for
    endif
    return
end
```

At each step, the next set of optimal 1-sequences of reversals is stored in a child node in lexicographic order. Among them, only those reversals are expanded that are *valid*. The reversals that are not *valid* are those that are either already processed or will be processed when they appear in the normal form order in some other branch of the tree. The normal form of a trace is the path from a reversal in the root node (level-1) to a reversal in the leaf node (level-d), such that all the reversals in the path are *valid*. Hence, the reversals that are expanded up to d-level correspond to the last reversal in the normal form of a trace whose subwords are stored in the string P. Therefore, when the routine *expand* is called and the last level d is reached then we print the normal form of the trace stored in P.

To implement this in parallel we distribute the 1-sequences of reversals stored in the root node (set R) among the parallel processes [12]. If the size of the set R is N and there are p number of processes then each process get N/p elements of R, if N is evenly divisible by p; otherwise the last process will get some extra elements. Now each process expands its given subset of elements and list the normal of traces obtained from them using the recursive routine *expand* (Algorithm 2) as described above.

B. Counting Total Number of Solutions

We extend the above algorithms (Algorithm 1 and Algorithm 2) to provide the count of number of solutions in the solution space. To accomplish this we have followed a greedy approach. We give preference to the commuting reversals by storing the set of next reversals in a particular order. First all the commuting reversals are stored in lexicographic order, followed by non-commuting reversals in lexicographic order. This allows us to append all the commuting reversals in lexicographic order to the current subword before the new subword begins. We append a non-commuting reversal (beginning of a new subword) only when there is no commuting reversal that can be appended next. This ensures that path leading to the normal of a trace is generated before any of its prefix is generated in other parts of the solution space. This helps in counting the total number of solutions

Same tree structure is used with slight modification. A node in the tree is represented by an array of the *rNode* structure with two additional fields (C and W):

1. *reversal* (R)
2. *permutation* (P)
3. *count* (C)- number of optimal 1-sequences of reversal

DOI: 01.IJIT.3.2.1228

obtained from *permutation*

4. *weight* (W)- number of solutions generated following the *reversals* in the path from the root node to the the current node

5. next

Next, we present the extended algorithms (Algorithm 3 and Algorithm 4).

Algorithm 3 *listTraces(* π*)*

```
begin
    d ← reversal distance of π
    R ← {ρ | ρ is an optimal 1-sequence for π in
                lexicographic order}
    N ← count(R)
    First ← allocate(N)
    W ← 0  [node weight]
    for j ← 0 to N-1 do
        [initialize rNode fields]
        initialize(First + j, ρ_j, π, 0, 0, null)
        [generate all the subsequent (d-1)-sequences]
        (First+j).w ← expand (First +j, d-1, ρ_j)
        W ← W + (First+j).w
    end for
    return W [Total number of solutions]
end
```

Algorithm 4 *expand*(current, d, P)

```
begin
    NW ← 0 [Node Weight]
    current.P ← applyReversal(current.R, current.P)
    if (d=0) then
      current.w ← 1
      current.next ← null
      NW ← 1
      printTrace(P)
    else
      R ← {ρ | ρ is an optimal 1-sequence for
                                  current.P}
     reorder(R, P) [arrange in the desired order]
      current.C ← count(R)
      B ← allocate(current.C)
      current.next ← B
      for j ← 0 to current.C-1 do
        [initialize rNode fields]
        initialize((B + j), ρ_j, current.P, 0, 0, null)
        [generate all the subsequent(d-1)
                    sequences]
                if valid(ρ_j, P) then
                    [append ρ_j to P and explore
    further]
                (B+j).w ← expand ((B +j), d-1, P.ρ_j)
        else
                    [insert ρ_j in nomal form order
    into P
                    and find its weight]
                (B+j).w ← findWeight(First, P
    * ρ_j)
                endif
        NW ← NW + (B+j).w
      end for
    endif
    return NW
end
```

The subroutine *reorder* arrange the next set of optimal 1-sequences of reversals in specific order, i.e. first all the commuting reversals arranged in lexicographic order then the non-commuting reversals in lexicographic order. Therefore, in the rearranged set of next reversals if any reversal is not *valid* then it must have been already processed. So, its weight is already calculated and we do not need to expand it further. We can find its weight by searching the tree. However, unlike Braga's algorithm, we do not have to search every time we add a reversal to the current trace. And the search space contains only those paths that precede the current trace P in normal form order. So it will be considerably faster. The field count (C) is useful for traversing the tree during searching. The advantage that we get by using depth first approach is reduced memory requirement. The total number of solutions is given by the node weight (sum of weights in each *rNode*) of the root node.

Note that many of the reversals that are present at the in a node are *redundant*. That is exploring them do not result in any new trace. The sooner we detect such cases the lesser will be computation and memory consumption. If we can identify such reversals at the first step then we can exclude them there itself. It will save lot of space and time as the number of prefixes to be computed is highly reduced. We know that two consecutive reversals α and β in an optimal sequence of reversals are commuting then the order in which these reversals are applied does not matter i.e., the sequence of reversals $\alpha\beta$ is equivalent to $\beta\alpha$. Therefore, if α and β are two commuting reversals in the root node such that $\{\{\beta\} \cup child(\beta)\} = \{\{\alpha\} \cup child(\alpha)\}$, where $child(\alpha)$ is the set of next reversals obtained after applying the reversal α, then all the traces generated from β will be same as those generated from α. If α precedes β in lexicographic order then we can discard β and we say that it is *duplicate* of α. We associate a flag E_α initialized to *true*, to determine if a reversal α in the root node is to be expanded or not.

In order to count the number of solutions correctly another flag D_α is used. It is initialized to 1 an incremented for each duplicate found. Hence, we have the following rule- if α and β are two reversals in the root node such that $commute(\alpha, \beta) = true$, $\alpha < \beta$, $E_\alpha = true$ and $\{\{\beta\} \cup child(\beta)\} = \{\{\alpha\} \cup child(\alpha)\}$ then E_β is *false* and increment D_α by one. A reversal α is expanded if the flag E_α is true and the total number of solutions returned by $expand(\alpha)$ is multiplied by D_α.

It is also possible to explore the traces in parallel. The reversals stored in the root node are distributed to parallel

DOI: 01.IJIT.3.2.1228

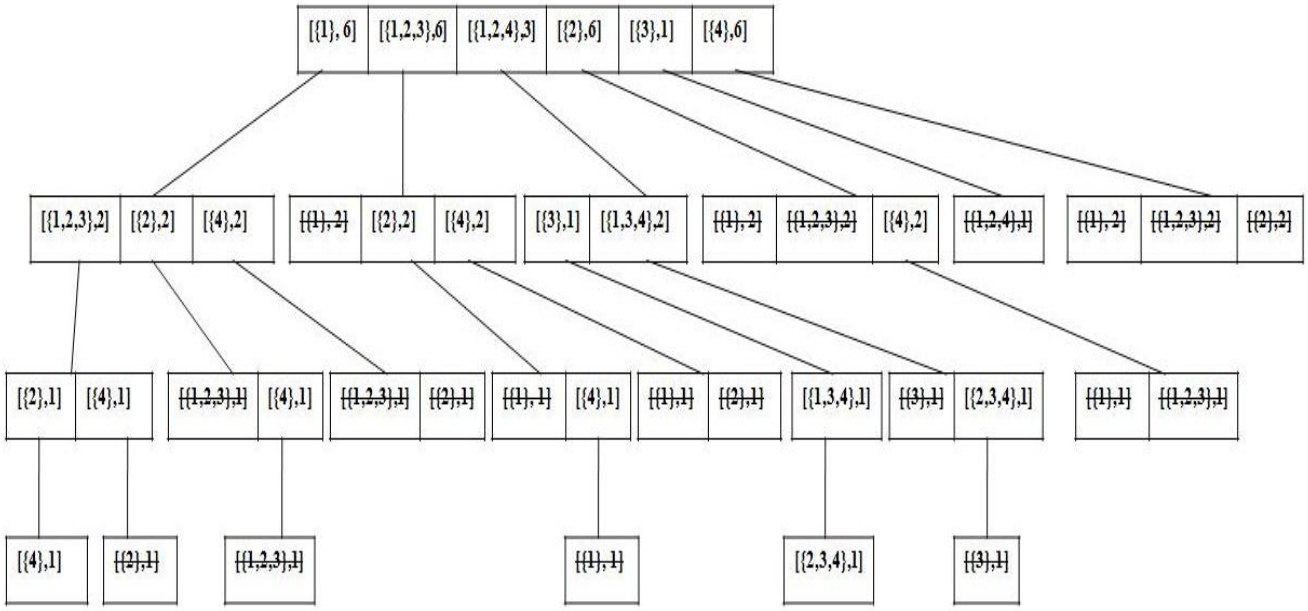

Figure 1. Tree structure of the traces that sort the permutation (-3 2 1 -4). In each *rNode* we have shown the corresponding reversal and the weight assigned to it. The values that are crossed belong to those *rNode*s that are not expanded further. When these reversals are added to their parent traces lead to previously identified traces

processes using the following rule: if α and β are two reversals in the root node such that *commute*(α, β) = *true*, α < β, and $\{\{\beta\} \cup child(\beta)\} \subset \{\{\alpha\} \cup child(\alpha)\}$ then both α and β must be given to the same process as the traces generated from β will be subset of those generated from α. Now each process can expand their set of reversals, independently, using the above approach. Finally, each process outputs the traces found by it and the total number of solutions in those traces.

IV. Results

We describe the result of applying the algorithms (Algorithm 3 and Algorithm 4) to an example permutation π = (-3 2 1 -4). The reversal distance (d) for this permutation is 4. The size of 1-sequence of reversals is 6, namely, {1}, {1, 2, 3}, {1, 2, 4}, {2}, {3} and {4}. These are stored in the *rNode* array of the root node in lexicographic order. The Algorithm 3 (*listTraces*) explores each of them one by one to identify the traces and count the number of solutions in each branch. The routine *expand* is called recursively for each *rNode*. It returns when either the last level d is reached or when the next reversal is not the last reversal when added to the current trace. To count the number of solutions originating from a *rNode*, the field weight (w) is used. All the weights are initialized to 0. Node weight (NW) is the sum of the individual weights in each *rNode* of that node. The weight (NW) is returned and assigned to the weight field of its parent *rNode*. If the next reversal is the last reversal when appended to the current trace then either one of the following two cases are possible: if d-level is not reached then it is explored further. Otherwise, if level-d is reached then both weight and node weight are assigned value 1(as there will be only element in this node). Otherwise if the next reversal is not the last reversal when appended to the current trace then its weight is searched in the tree by matching the sequence of reversals stored in the current prefix (P). The final output is shown in Fig. 1. The node weight of the root node (6+6+3+6+1+6 = 28) gives the total number of solutions for the example permutation π.

Conclusions

We have shown that it is possible to count the total number of solutions even if we proceed in depth first manner. Using depth first approach not only reduces memory consumption but also makes it easier to parallelize. This allows us to efficiently handle large permutations. Also, by giving preference to the commuting reversals over non-commuting reversals and excluding *redundant* reversals at every step we can develop an algorithm to directly enumerate the normal of traces. Another future scope is to extend the proposed method to compute the number of solutions in each trace as well.

References

[1] S. Hannenhali, and P. A. Pevzner, "Transforming Cabbage into Turnip," in 27th ACM-SIAM Symposium on Theory of Computing, pp. 178—189, Las Vegas, USA (1995).

[2] E. Tannier, A. Bergeron, and M. F. Sagot, "Advances on sorting by reversals," in Disc. Appl. Math., Elsevier 155(6-7), pp. 881—888 (2007).

[3] D. A. Bader, B. M. Moret, and M. Yan, "A Linear-Time Algorithm for Computing Inversion Distance between Signed Permutations with an Experimental Study," in J. Comput. Biol. 8(5), pp. 483—491(2001).

[4] Y. Ajana, J.-F. Lefebvre, E. R. M. Tillier, and N. El-Mabrouk, "Exploring the set of all minimal sequences of reversals – an application to test the replication-directed reversal hypothesis," in Guigo, R., Gusfield, D. (eds.) WABI 2002. LNCS, vol. 2452, pp. 300—315. Springer, Heidelberg (2002).

[5] A. C. Siepel, "An algorithm to find all sorting reversals," in 6th

DOI: 01.IJIT.3.2.1228

annual International Conf. on Comput. Mol. Biol. (RECOMB 2002), pp. 281—290. ACM Press, New York (2002).

[6] K. M. Swenson, G. Bader, and D. Sankoff, "Listing all sorting reversals in quadratic time," in: Singh, M. (eds.) WABI 2010, LNCS, vol. 6293, pp. 102—110. Springer, Heidelberg (2010).

[7] A. Bergeron, C. Chauve, T. Hartman, and K. Saint-Onge, "On the properties of sequences of reversals that sort a signed permutation," in: JOBIM, pp. 99—108 (2002).

[8] V. Diekert, and G. Rozenberg, "The Book of Traces," in World Scientific, Singapore (1995).

[9] M. D. V. Braga, "Baobabluna: the solution space of sorting by reversals," in Bioinformatics 25 (14), pp. 1833—35 (2009).

[10] M. D. V. Braga, M. Sagot, C. Scornavacca, and E. Tannier, "Exploring the Solution Space of Sorting by Reversals, with Experiments and an Application to Evolution," in IEEE/ACM Transactions on Computational Biology 5(3), pp. 348—356 (2008).

[11] C. Baudet, and Z. Dias, "An improved algorithm to enumerate all traces that sort a signed permutation by reversals," in 2010 ACM Symposium on Applied Computing, pp. 1521—1525 (2010).

[12] Amritanjali, and G. Sahoo, "Parallel strategy for exploring the solution space for sorting by reversals," ARTCom 2012, LNEE pp. 110–117, Springer-Verlag Berlin Heidelberg 2012.

DOI: 01.IJIT.3.2.1228